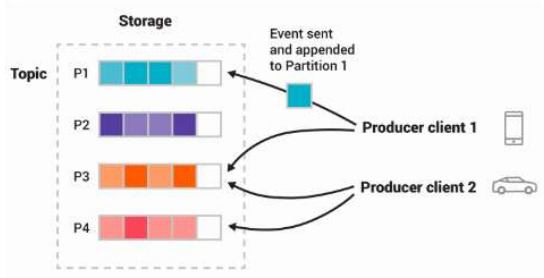
Exhibit 7

Exhibit 7 to Complaint
Intellectual Ventures I LLC and Intellectual Ventures II LLC

Example Comerica Count II Systems and Services
U.S. Patent No. 8,407,722 ("the '722 Patent")

The Accused Systems and Services include, without limitation, Comerica's systems that utilize Kafka ("Kafka"); all past, current, and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current and future Comerica's systems and services that have the same or substantially similar features as the specifically identified systems and services ("Example Comerica Count II Systems and Services").

U.S. Patent No. 8,407,722	
Example Claim 14	**Example Comerica Count II Systems and Services**
14. A method comprising: providing, using a processing device of an input source, a data representation to a client device, different from the input source, coupled to a routing network, wherein the data representation includes at least one live object recognizable by the client device, and causing the client device to respond to the live object of the data representation by determining an object identifier (ID) of the live object and to register for updates of the live object with the routing network, such that registering the client device with the routing network provides client connection information to a node in the routing network; and	Upon information and belief, Comerica's systems perform "providing, using a processing device of an input source, a data representation to a client device, different from the input source, coupled to a routing network, wherein the data representation includes at least one live object recognizable by the client device, and causing the client device to respond to the live object of the data representation by determining an object identifier (ID) of the live object and to register for updates of the live object with the routing network, such that registering the client device with the routing network provides client connection information to a node in the routing network." Kafka facilitates the receipt of messages identifying a live object and containing data for updating the property of the live object at end-user devices. **How does Kafka work in a nutshell?** Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments. **Servers**: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss. **Clients**: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs. *See* https://kafka.apache.org/documentation/ (last accessed on November 10, 2023).

	Kafka clusters receive an update message sent by a input source. **2.1 Producer API** The Producer API allows applications to send streams of data to topics in the Kafka cluster. *See* https://kafka.apache.org/documentation/ (last accessed on November 10, 2023). **Producers** are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.  *See* https://kafka.apache.org/intro (last accessed on November 10, 2023). Comerica implements Kafka to provide end users with updates in real time. *See* https://www.jobzmall.com/comerica-bank/job/mobile-application-developer-7 (last accessed on November 11, 2023).

	See https://apps.apple.com/us/app/comerica-mobile-banking/id403598968 (last accessed on November 11, 2023).
	
	See https://apps.apple.com/us/app/comerica-treasury-mobile/id1051410838?platform=iphone (last accessed on November 11, 2023).
	
	Each new event/record updates a previous record.
	
	Kafka is a distributed *event streaming platform* that lets you read, write, store, and process *events* (also called *records* or *messages* in the documentation) across many machines.
	
	Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in *topics*. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder.
	
	See https://kafka.apache.org/documentation/#producerapi (last accessed on November 10, 2023).
	
	send
	
	`public Future<RecordMetadata> send(ProducerRecord<K,V> record)`
	
	Asynchronously send a record to a topic. Equivalent to `send(record, null)`. See `send(ProducerRecord, Callback)` for details.
	
	Specified by:
	
	`send` in interface `Producer<K,V>`
	
	Parameters:
	
	`record` - The record to send
	
	Returns:
	
	A future which will eventually contain the response information

	See https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html (last accessed on November 10, 2023). Each new event/record updates a previous record. `KTable` is an abstraction of a *changelog stream* from a primary-keyed table. Each record in this changelog stream is an update on the primary-keyed table with the record key as the primary key. *See* https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/KTable.html (last accessed on November 10, 2023). Let's illustrate this with an example. Imagine a table that tracks the total number of pageviews by user (first column of diagram below). Over time, whenever a new pageview event is processed, the state of the table is updated accordingly. Here, the state changes between different points in time – and different revisions of the table – can be represented as a changelog stream (second column). *See* https://docs.confluent.io/platform/current/streams/concepts.html (last accessed on November 10, 2023).
sending, using the processing device of the input source, an update message to the routing network, wherein the update message identifies the live object and contains update data that updates a property of the live object;	Upon information and belief, Comerica's systems perform "sending, using the processing device of the input source, an update message to the routing network, wherein the update message identifies the live object and contains update data that updates a property of the live object." Each new event/record comprises an event key. The key is used to determine the partition/topic ("a category") to which the event should be written.

send

`public Future<RecordMetadata> send(ProducerRecord<K,V> record)`

Asynchronously send a record to a topic. Equivalent to `send(record, null)`. See `send(ProducerRecord, Callback)` for details.

Specified by:

`send` in interface `Producer<K,V>`

Parameters:

`record` - The record to send

Returns:

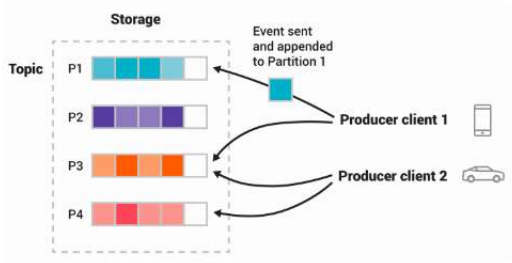
A future which will eventually contain the response information

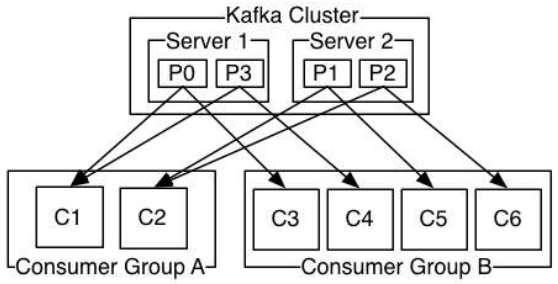
See https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html (last accessed on November 10, 2023).

Each new event/record comprises an event key. The key is used to determine the partition/topic ("a category") to which the event should be written.

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

	Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.  *Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.* *See* https://kafka.apache.org/ (last accessed on November 10, 2023).
wherein a gateway device at the routing network is configured to identify a category of the update message based on the input source, to determine a node type to which the identified	Upon information and belief, Comerica's systems include "wherein a gateway device at the routing network is configured to identify a category of the update message based on the input source, to determine a node type to which the identified category maps, and to route the update message to the node, having the node type, at the routing network." Comerica includes Kafka Clusters that comprise multiple Kafka brokers ("nodes"). A broker includes multiple topics (and partitions). A broker is selected based upon the topic/partition to which the event belongs. The broker stores the events inside the partitions of the topics.

category maps, and to route the update message to the node, having the node type, at the routing network,;	**Servers**: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.  A two server Kafka cluster hosting four partitions (P0-P3) with two consumer groups. Consumer group A has two consumer instances and group B has four.

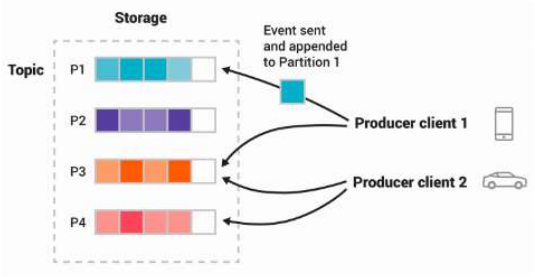


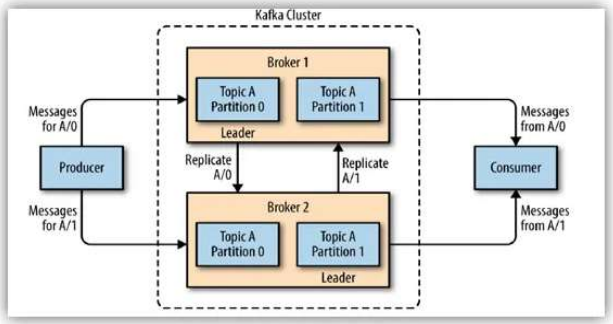
Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

See https://kafka.apache.org/documentation/ (last accessed on November 10, 2023).

A Kafka Cluster comprises of multiple Kafka brokers ("nodes"). A broker includes multiple topics (and partitions). A broker is selected based upon the topic/partition to which the event belongs. The broker stores the events inside the partitions of the topics.

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

	Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written. *See* https://kafka.apache.org/ (last accessed on November 10, 2023).  *See* Kafka Architecture & Internal. This talks about the Apache Kafka… | by Narayan Kumar | Medium (last accessed on November 11, 2023).
wherein the node is configured to identify the client device as a registered device and to route the update message to the client device, and	Upon information and belief, Comerica's systems include "wherein the node is configured to identify the client device as a registered device and to route the update message to the client device." Consumers subscribe ("register") to a topic (all the events present inside the topic). The broker pushes ("routes") the events of a topic to the consumer in real time that have subscribed to the topic.

Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be "payments". Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.

See https://kafka.apache.org/intro (last accessed on November 10, 2023).

Consumers subscribe ("register") to a topic (all the events present inside the topic). The broker pushes ("routes") the events of a topic to the consumer in real time that have subscribed to the topic.

	Push vs. pull An initial question we considered is whether consumers should pull data from brokers or brokers should push data to the consumer. In this respect Kafka follows a more traditional design, shared by most messaging systems, where data is pushed to the broker from the producer and pulled from the broker by the consumer. Some logging-centric systems, such as **2.4 Connect API** The Connect API allows implementing connectors that continually pull from some source data system into Kafka or push from Kafka into some sink data system. Many users of Connect won't need to use this API directly, though, they can use pre-built connectors without needing to write any code. Additional information on using Connect is available here. *See* https://kafka.apache.org/documentation/ (last accessed on November 10, 2023).
wherein the client device processes the update message upon receipt to update the property of the live object at the client device.	Upon information and belief, Comerica's systems include "wherein the client device processes the update message upon receipt to update the property of the live object at the client device." Upon receiving the events from one or more topics, the consumer processes the events. Processing might be an aggregation of events into a new topic.

Kafka APIs

In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The Admin API to manage and inspect topics, brokers, and other Kafka objects.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.

Stream Processing

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called Kafka Streams is available in Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include Apache Storm and Apache Samza.

See https://docs.Kafka.com/storage/storagedriver/ (last accessed on November 11, 2023).